(binary) MAX-heap

16
  14
   8
  14
   7
  10
   9
   3

2
  4
  1
(binary) Max-heap

Rules:
- last row filled from left
- other rows full
- parent > children
Binary MAX-heap

Rules:
- last row filled from left
- other rows full
- parent > children

Notice every subtree is also a heap
(binary) MAX-heap

Rules:
- last row filled from left
- other rows full
- parent > children

Notice every subtree is also a heap
(binary) \textit{MAX-heap}

Rules:
- last row filled from left
- other rows full
- parent > children

\begin{align*}
\text{left-child}(i) &= 2i \\
\text{right-child}(i) &= 2i + 1
\end{align*}

[Notice every subtree is also a heap]
(binary) Max-heap

Rules:
- last row filled from left
- other rows full
- parent > children

\[
\text{parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor
\]

\[
\text{left-child}(i) = 2i
\]

\[
\text{right-child}(i) = 2i + 1
\]
(binary) Max-heap

Rules:
- Last row filled from left
- Other rows full
- Parent > children

\[
\text{parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor
\]

\[
\text{left-child}(i) = 2i
\]

\[
\text{right-child}(i) = 2i + 1
\]

[Notice every subtree is also a heap]

1 2 3 4 5 6 7 8 9 10

16 14 10 8 7 9 3 2 4 1
How does this relate to sorting?
How does this relate to sorting?

Largest element is on top.
How does this relate to sorting?

Largest element is on top.
2nd largest is in level 2.
How does this relate to sorting?

Largest element is on top.
2nd largest is in level 2.
3rd largest is 4 in level 2.
How does this relate to sorting?

Largest element is on top.
2nd largest is in level 2.
3rd largest is
4 in level 2
or
4 in level 3
How does this relate to sorting?

Largest element is on top.

2nd largest is in level 2.

3rd largest is
  1) in level 2
     OR
  2) in level 3
     & child of 2nd
How does this relate to sorting?

Largest element is on top.
2nd largest is in level 2.
3rd largest is in level 2 or level 3 & child of 2nd.

getting messy
attempt to extract MAX & restore heap
attempt to extract MAX & restore heap
attempt to extract MAX & restore heap
attempt to extract MAX & restore heap
attempt to extract MAX & restore heap
14
  /  \\  
  8   10
   /  \\  
  4   7   9
     /  \\  
    2   1   3

attempt to extract MAX & restore heap

now what? .... failed
attempt to extract MAX & restore heap

if we can do this in $O(\log n)$ time, we have a sorting algorithm
extract MAX
- extract MAX
- place rightmost leaf at top
(strange move... it's small!)
- extract MAX
- place rightmost leaf at top
  (strange move... it's small!)
- extract MAX
- place rightmost leaf at top
  (strange move... it's small!)
- swap top w/ largest child
  (locally restore parent > child)
- extract MAX
- place rightmost leaf at top
  (strange move... it's small!)
- swap top w/ largest child
  (locally restore parent>child)
- extract MAX
- place rightmost leaf at top
  (strange move... it's small!)
- swap top w/ largest child
  (locally restore parent>child)
- repeat downward while needed
- extract MAX
- place rightmost leaf at top
  (strange move... it's small!)
- swap top w/ largest child
  (locally restore parent's child)
- repeat downward while needed
- extract MAX
- place rightmost leaf at top
  (strange move... it's small!)
- swap top w/ largest child
  (locally restore parent's child)
- repeat downward while needed
• extract MAX

• place rightmost leaf at top
  (strange move... it's small!)

• swap top w/ largest child
  (locally restore parent>child)

• repeat downward while needed
- extract MAX
- place rightmost leaf at top
  (strange move... it's small!)
- swap top w/ largest child
  (locally restore parent>child)
- repeat downward while needed

\[ \text{time?} \]
height of heap?

- extract MAX
- place rightmost leaf at top (strange move... it's small!)
- swap top w/ largest child (locally restore parent > child)
- repeat downward while needed
time?
height of heap? → Θ(log n)

- extract MAX
- place rightmost leaf at top (strange move... it's small!)
- swap top w/ largest child (locally restore parent > child)
- repeat downward while needed

time?
height of heap? $\rightarrow \Theta(\log n)$  

Ready to extract new MAX

- extract MAX
- place rightmost leaf at top (strange move... it's small!)
- swap top w/ largest child (locally restore parent>child)
- repeat downward while needed

$\rightarrow$ time?
To work in-place when extracting max we can swap it with leaf.
To work in-place when extracting max we can swap it w/ leaf.

Instead of deleting this node, just ignore it.

Notice max is stored at the max index of our array.
To work in-place when extracting max we can swap it w/ leaf.
To work in-place when extracting MAX we can swap it w/ leaf.
To work in-place when extracting MAX we can swap it w/ leaf.
To work in-place when extracting \text{MAX} we can swap it w/ leaf.
To work in-place when extracting max we can swap it w/ leaf.
To work in-place when extracting MAX we can swap it w/ leaf.
To work in-place when extracting MAX we can swap it w/ leaf.
To work in-place when extracting MAX we can swap it w/ leaf.
To work in-place when extracting \( \text{MAX} \) we can swap it with a leaf.
To work in-place when extracting MAX we can swap it w/ leaf.
To work in-place when extracting max we can swap it with leaf.
To work in-place when extracting MAX we can swap it with leaf.
To work in-place when extracting max we can swap it w/ leaf.
To work in-place when extracting MAX we can swap it w/ leaf.
To work in-place when extracting MAX we can swap it w/ leaf.
To work in-place when extracting max we can swap it w/ leaf.
To work in-place when extracting max we can swap it w/ leaf.
To work in-place when extracting MAX we can swap it w/ leaf.
To work in-place when extracting max we can swap it w/ leaf.
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

\[ \text{do this } n \text{ times} \rightarrow O(n \log n) \text{ sorting} \]

But how did we have a heap in the first place?
So we can extract $\text{MAX}$ & maintain a heap, in $O(\log n)$ time.

Do this $n$ times $\Rightarrow O(n \log n)$ sorting

But how did we have a heap in the first place?

Start with unsorted elements
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

\[ \text{do this } n \text{ times } \rightarrow O(n \log n) \text{ sorting} \]

But how did we have a heap in the first place?

\[
\begin{array}{ccc}
2 & 3 & 10 \\
4 & 7 & 9 \\
16 & 14 & 1 \\
\end{array}
\]

make heap of size 1
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

$\downarrow$ do this $n$ times $\rightarrow O(n \log n)$ sorting

But how did we have a heap in the first place?
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

Do this $n$ times → $O(n \log n)$ sorting

But how did we have a heap in the first place?

![Diagram of a heap with numbers 2, 3, 4, 7, 8, 10, 14, 9, 1 with arrows indicating the process of extracting MAX and maintaining the heap property.](attachment:heap-diagram.png)

trivially get heap of size 2, possibly w/ a swap.
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

Do this $n$ times $\rightarrow O(n \log n)$ sorting

But how did we have a heap in the first place?

generally, insert new element as rightmost leaf in lowest level
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

**Do this $n$ times $\rightarrow O(n \log n)$ sorting**

But how did we have a heap in the first place?

Then repeat swapping while required.
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

Do this $n$ times $\Rightarrow O(n \log n)$ sorting

But how did we have a heap in the first place?

then repeat swapping while required
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

Do this $n$ times $\rightarrow O(n\log n)$ sorting

But how did we have a heap in the first place?
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

Do this $n$ times $\rightarrow O(n \log n)$ sorting

But how did we have a heap in the first place?
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

Do this $n$ times $\Rightarrow O(n \log n)$ sorting

But how did we have a heap in the first place?
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

Do this $n$ times $\rightarrow O(n \log n)$ sorting

But how did we have a heap in the first place?

[Diagram of a heap with nodes labeled 2, 4, 10, 7, 8 and 1, 3, 9, 16, 14.]
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

\[ \text{do this } n \text{ times } \rightarrow O(n \log n) \text{ sorting} \]

But how did we have a heap in the first place?
When inserting a new leaf (wlog R) there is a problem iff $R > P$. 
When inserting a new leaf (wlog $R$) there is a problem iff $R > P$.

By swapping $R \leftrightarrow P$ we have a heap in $\text{subtree}(R)$ ($R > P > L$).
When inserting a new leaf (wlog $R$) there is a problem iff $R > P$.

By swapping $R \leftrightarrow P$ we have a heap in $\text{subtree}(R)$ ($R > P > L$).

...but we may have a new problem iff $R > G$. 


When inserting a new leaf (wlog $R$) there is a problem iff $R > P$.

By swapping $R \leftrightarrow P$ we have a heap in subtree($R$) $(R > P > L)$.

...but we may have a new problem iff $R > G$ then $R \leftrightarrow G$. $G > P > L : \text{OK}$ & $R > G > X : \text{OK}$.
R will move up some path until smaller than node above.
R will move up some path until smaller than node above. That path will "shift down" so every subtree has a larger root.

A: don't care
R will move up some path until smaller than node above.

That path will "shift down" so every subtree has a larger root.

\[ O(n \log n) \text{ time per node} \]

and in-place: iterate on array. Only swaps used.
(another) In-place heap-build
(another) In-place heap-build

Iterate from end of array

\[ \text{from right to left, on each level, starting at bottom} \]
(another) In-place heap-build

Iterate from end of array from right to left
on each level, starting at bottom

heapify each node at \( x \), i.e. subtree at \( x \).
(another) In-place heap-build

Iterate from end of array from right to left on each level, starting at bottom

heapify each node $i$, i.e., subtree at $x$. 

already heaps
Heapify $X$:

"Can assume each child is a heap."
Heapify $X$:

// can assume each child is a heap.

Might have to swap $X$ with one of its children & further down levels

already heaps
Heapify $x$:

- Can assume each child is a heap.

- Might have to swap $x$ with one of its children and further down levels.
Heapify $x$:

- Can assume each child is a heap.
- Might have to swap $x$ with one of its children & further down levels.

Time for $x = O(\text{height}(x))$

Overall $O(n \log n)$
Time = $O(h(x))$
Time = \( O(h(x)) \)

Total time: \( O(\sum h(x)) \)
Time = O(h(x))

Total time: O(\sum_{all x} h(x))

\[ \sum \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \cdots + 2 \cdot ((\log n) - 1) + 1 \cdot \log n \]
\[ \sum \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \ldots + 2 \cdot ((\log n) - 1) + 1 \cdot \log n \]

\[ = \sum_{h=1}^{\log n} \frac{n}{2^h} \cdot h \]
$$\sum \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \ldots + 2 \cdot ((\log n)-1) + 1 \cdot \log n$$

$$= \sum_{h=1}^{\log n} \frac{n}{2^h} \cdot h = n \cdot \sum \frac{h}{2^h}$$

Time = $O(h(x))$

Total time: $O\left(\sum_{x} h(x)\right)$
\[ \sum \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \ldots + 2 \cdot ((\log n) - 1) + 1 \cdot \log n \]

\[ = \sum_{h=1}^{\log n} \frac{n}{2^h} \cdot h = n \cdot \sum \frac{h}{2^h} \leq n \cdot \frac{1/2}{(1 - 1/2)^2} \]

\[ \text{Time} = O(h(x)) \]

\[ \text{Total time: } O\left(\sum_{\text{all } x} h(x)\right) \]

\[ \left[ \begin{array}{l}
\text{CLRS 11.48}
\text{use } \sum_{k=0}^{\infty} k x^k
\end{array} \right] \]
\[ \sum \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \cdots + 2 \cdot ((\log n) - 1) + 1 \cdot \log n \]

\[ = \sum_{h=1}^{\log n} \frac{n}{2^h} \cdot h = n \cdot \sum \frac{h}{2^h} \leq n \cdot \frac{1/2}{(1-1/2)^2} = O(n) \]